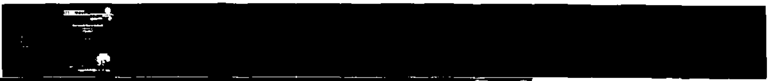
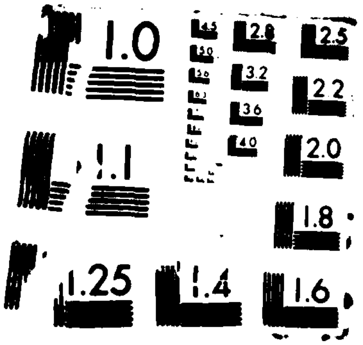
AD-A182 130 SOME COMPLEXITY THEORY FOR CRYPTOGRAPHY(U) NAVAL RESEARCH LAB WASHINGTON DC A M GAGLIONE 12 JUN 87 NRL-9024 1/1

UNCLASSIFIED F/G 12/9 NL



END
DATE
FILMED
8


1.0
4.5
5.0
5.6
6.3
2.8
2.5
3.2
2.2
3.6
4.0
2.0
1.1
1.8
1.25
1.4
1.6

A182130

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION
UNCLASSIFIED

1b. RESTRICTIVE MARKINGS

2a. SECURITY CLASSIFICATION AUTHORITY

2b. DECLASSIFICATION / DOWNGRADING SCHEDULE

3. DISTRIBUTION / AVAILABILITY OF REPORT
Approved for public release; distribution unlimited.

4. PERFORMING ORGANIZATION REPORT NUMBER(S)
NRL Report 9024

5. MONITORING ORGANIZATION REPORT NUMBER(S)

6a. NAME OF PERFORMING ORGANIZATION
Naval Research Laboratory

6b. OFFICE SYMBOL (If applicable)
Code 5350

7a. NAME OF MONITORING ORGANIZATION

6c. ADDRESS (City, State, and ZIP Code)
Washington, DC 20375-5000

7b. ADDRESS (City, State, and ZIP Code)

8a. NAME OF FUNDING / SPONSORING ORGANIZATION
Naval Air Systems Command

8b. OFFICE SYMBOL (If applicable)
APC-209C

9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER

8c. ADDRESS (City, State, and ZIP Code)
Washington, DC 20361

10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO	WORK UNIT ACCESSION NO.
64211N			DN180-248

11. TITLE (Include Security Classification)
Some Complexity Theory for Cryptography

12. PERSONAL AUTHOR(S)
Gaglione, Anthony M.

13a. TYPE OF REPORT
Final

13b. TIME COVERED
FROM 6/85 TO 8/85

14. DATE OF REPORT (Year, Month, Day)
1987 June 12

15. PAGE COUNT
22

16. SUPPLEMENTARY NOTATION
Dr. Gaglione's permanent address is Department of Mathematics, U.S. Naval Academy, Annapolis, MD 21402-5000.

17. COSATI CODES

FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
Cryptography
Complexity theory
Public key codes
Turing machines

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

This report concerns some of the elementary concepts in complexity theory. In particular, a mathematical model is developed for a finite-state machine and the Turing machine. This model has applications to public key cryptosystems, in determining which problems are P, NP, or NPC. The report was written to be as accessible to the nonspecialist as possible.

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS

21. ABSTRACT SECURITY CLASSIFICATION
UNCLASSIFIED

22a. NAME OF RESPONSIBLE INDIVIDUAL
E. Vegh

22b. TELEPHONE (Include Area Code)
(202) 767-3481

22c. OFFICE SYMBOL
Code 5350

83 APR edition may be used until exhausted
All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE

U.S. Government Printing Office: 1985—507-047

CONTENTS

SOME COMPLEXITY THEORY FOR CRYPTOGRAPHY

INTRODUCTION

The cryptographic strength of public key cryptosystems usually depends on the underlying assumption that certain known mathematical problems are difficult to solve. For example, we shall see that the RSA system (named after its inventors: Rivest, Shamir, and Adleman [1]) is a cryptosystem whose "breaking" can depend on the solution of a "hard" mathematical problem, i.e., the factoring problem. Thus, it becomes significant for cryptography to classify in some way those mathematical problems that are "hard." Complexity theory attempts to do this. This report presents a short introduction to complexity theory. The motivation for this study is its usefulness in cryptography.

This report also presents an adequate model, or simulator, of an algorithm or effective procedure—the Turing machine. We also discuss a universal simulator for all such machines (the theoretical inspiration for the stored-program computer). We find there are problems for which no algorithmic solution can ever be found. Finally, for problems which have solution algorithms we discuss a way to measure the relative efficiency of these algorithms.

THE RSA SYSTEM

Conventional cryptologic systems have the disadvantage that, for efficient decoding, a key to the decoding process must be separately and securely passed to the receiver. A public key cryptosystem, on the other hand allows any participant P in a communications network to publicize his encoding scheme to the network. Doing so, however, does not disclose the key to the decoding process. Any member, S, of the network who wishes to send a secret communication to P may do so by encoding it with P's known coding scheme, and only P will be able to decode it. Secure communication of a decoding key is not necessary. It is also desirable that the method have an authentication feature; i.e., S can encode the message to P in such a way as to include S's "signature." A signature is some proof that this particular message came from S.

One elegant candidate for such a scheme is the RSA system. To describe the RSA algorithm, we need some notation and some elementary number theory [2].

Let $N = \{1, 2, 3, \ldots m, \ldots\}$ be the set of natural numbers. For $m \in N$ let $\phi(m)$ be Euler's ϕ-function of m where $\phi(m)$ = the number of natural numbers, k, such that $k < m$ and the greatest common divisor of k and m is 1 (denoted here by greatest common denominator (gcd) $(k,m) = 1$). For a, b, any integers with m, a positive integer greater than 1, we write the expression

$$a \equiv b(\bmod m),$$

(read "a is congruent to b modulo m") to denote the fact that the integer m exactly divides $a - b$. The RSA technique makes use of the following simple number theoretical result:

Euler's Theorem: If $a, m \in N$ with gcd $(a,m) = 1$, then $a^{\phi(m)} \equiv 1 \bmod m$.

The method works as follows. Two large prime numbers p and q (about 100 digits each) are chosen at random, and $p \cdot q = n$ is computed. Letting $m = \phi(n) = (p-1)(q-1)$, a large random

Manuscript approved September 26, 1986.

number y is chosen such that gcd $(y,m) = 1$. This step guarantees the existence of an integer $x, 0 < x < m$, such that

$$x \cdot y \equiv 1(\bmod m).$$

Efficient computer algorithms exist for producing p, q, y, and x. The message to be encoded is translated into a string of integers in the set $\{0, 1, 2, \ldots, 26\}$; e.g., blank $= 0$, $A = 1$, $B = 2, \ldots Z = 26$. The resulting string is then treated as a single number T, $0 \leq T \leq n - 1$, or as a sequence of such numbers. The enciphering process, finding $E_{K_1}(T)$, consists of computing

$$E_{K_1}(T) \equiv T^y \ (\bmod n)$$

where $0 \leq E_{K_1}(T) \leq n - 1$. The encryption scheme is made public by announcing n and y. Using common terminology, the public key is

$$K_1 = \{n, y\}.$$

To decode the ciphertext $C = E_{K_1}(T)$, one computes

$$D_{K_2}(T) = C^x \equiv T^{xy} \ (\bmod n),$$

where $0 \leq C^x \leq n - 1$. We note that

$$T^{xy} = T^{1+k\phi(n)} \equiv T(\bmod n)$$

by Euler's theorem. Since $T < n$, $C^x = T$ is now uniquely determined. The key to the decoding, x, is not made public. Thus, the secret decryption key is

$$K_2 = x.$$

Both the enciphering and deciphering processes can be done efficiently by computer. Breaking the algorithm, however, requires finding the prime factors p and q of n which, at present, cannot be done efficiently. If n is a 200-digit number for example, it is estimated that finding its prime factors, by using a high-speed computer and the best factoring algorithms currently known, could require about 3 billion years.

Before getting involved in our discussion of complexity theory, we make the following caveat. Although some mathematical problem may prove to be "hard," it is not true that the cryptosystem which is based on this "hard" problem will be as hard to break. We discuss this later in this report.

FINITE-STATE MACHINES

In this section, we consider an attempt to simulate general computation. In fact, a finite-state machine simulates computational devices such as modern digital computers. We see how adequate this structure is as a model of computation in the most general sense by characterizing its capabilities as a "recognizer." To attempt to build a mathematical model describing finite-state machines, we first try to abstract some of the important features.

- Operations of the machine are *synchronized*. We only look at the machine at fixed times, or clock pulses, t_0, t_1, t_2, etc. We assume the machine is discrete so that the responses to an input at t_i appear at t_{i+1}.

- The machine is *deterministic*; i.e., its actions in response to a given sequence of inputs are completely predictable.

- The machine responds to *inputs*.

- There is a finite number of *states* the machine can attain. At any time t_i the machine is in exactly one of these states. Which state it will be in at t_{i+1} is a function of both its present state and present input. The present state, however, depends upon the previous state and input and so forth back to the initial operation. Thus, the state of the machine at any moment serves as a form of memory of past inputs.

- The machine is capable of *output*. The nature of the output is a function of the present state of the machine. Thus it also depends upon past inputs.

A modern digital computer has these five features. Its operations are synchronized by clock pulses (although very rapid); it operates in a deterministic fashion and is capable of responding to inputs. A computer is composed of a large number of bistable "on-off" elements. If there are n such elements, there are altogether 2^n on-off configurations which the computer can be in. These configurations are the states of the computer, and this number is finite (although very large). The present state of the computer (the present memory configuration) reflects its history of past states and inputs. Finally, the output at any moment depends upon the present state of the machine.

We are now ready for the formal definition:

Definition 1 — $M = [S, I, O, f_s, f_O]$ is a *finite-state machine* if S is a finite set of states, I is a finite set of input symbols (the input alphabet), O is a finite set of output symbols (the output alphabet), and f_s and f_O are functions where, f_s: $S \times I \rightarrow S$ and $f_O : S \rightarrow O$. The machine is always initialized to begin in a fixed starting state, called s_O here.

The function f_s is the next-state function. It maps a (state, input) pair to a state. Thus, the state at clock pulse t_{i+1}; state (t_{i+1}), is obtained as follows:

$$\text{state } (t_{i+1}) = f_s(\text{ state } (t_i), \text{ input } (t_i)).$$

The function f_O is the output function. When f_O is applied to a state at time t_i, we get

$$\text{output } (t_i) = f_O(\text{ state } (t_i)).$$

Notice that the effect of applying function f_O is available instantly, but the effect of applying f_s is not available until the next clock pulse. To describe a finite-state machine, we can use either of two alternatives: (a) The *state table* actually lists sets S, I, and O and tabulates the functions f_s and f_O. (b) The *state graph*, a directed graph, has each state of M with its corresponding output as vertices, and the next-state function is given by directed edges of the graph with each edge showing the input symbol(s) that produces that particular state change.

To illustrate state tables and graphs, we give some simple examples.

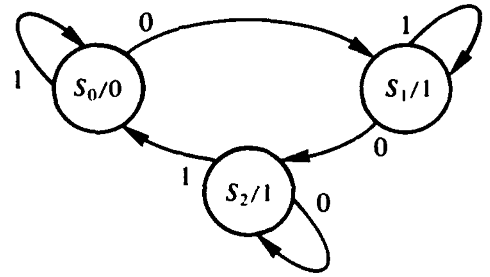
Example 1 — Let M be a machine with $S = \{s_0, s_1, s_2\}$, $I = O = \{0, 1\}$, and f_s and f_O defined by the following state table:

Present State	Next State		Output
	Present Input		
	0	1	
s_0	s_1	s_0	0
s_1	s_2	s_1	1
s_2	s_2	s_0	1

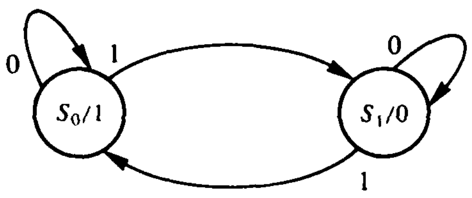
The machine M begins in state s_o which has an output of 0. If the first input symbol is a 0, the next state of the machine, $f_s(s_0,0) = s_1$. S_1 has an output, $f_O(s_1) = 1$. If the next input symbol is 1, the machine stays in state s_1, $f_s(s_1, 1) = s_1$, with output 1. Continuing this procedure, an input sequence 01101 (read left to right) would produce the following:

Time	t_0	t_1	t_2	t_3	t_4	t_5
Input	0	1	1	0	1	—
State	s_0	s_1	s_1	s_1	s_2	s_0
Output	0	1	1	1	1	0

The initial 0 of the output string is spurious—it merely reflects the starting state, not the result of any input. The state graph of M is given as follows:



Example 2 — The machine M described here is a parity-check machine. When the input received through time t_i contains an even number of 1s, then the output at time t_{i+1} is 1; otherwise, the output is 0. The state graph of M is given as follows:



For simplicity, we assume that our machines have the same input and output alphabet, usually $I = O = \{0,1\}$. Also, we denote by I^* and O^* the sets of all strings of elements of I and O, respectively (here we include the empty set, ϕ). Example 2 already exhibits a finite-state machine acting as a "recognizer." This recognizer signals with an output of 1 whenever an input string belonging to a particular set of possible input strings has been received. The machine of Example 2 recognizes the set of all strings consisting of an even number of 1s.

Now, we want to see precisely which sets the finite-state machines are capable of recognizing. Recognition is possible because machine states can have a limited memory of past inputs. Even though the machine is finite, it is possible for a particular input signal to affect the behavior of a machine "forever." However, not every input signal can do this and there are some classes of inputs that require remembering so much information that no machine can detect them.

To avoid writing down outputs, we designate those states of a finite-state machine with output 1 as *final states* and use a double circle to denote then in the state graph. Thus we give the following definition of recognition:

Definition 2 — A finite-state machine M with input alphabet I recognizes a subset S of I^* if M, beginning in state s_0 and processing an input string α, ends in a final state if and only if $\alpha \in S$.

We next introduce compact symbolism to describe the sets of interest to us. We describe these sets by using "regular expressions"; each regular expression describes a particular set. First, we define what regular expressions are; then we see how a regular expression describes a set. We assume here that I is some finite set of symbols; later, I will be the input alphabet for a finite-state machine.

Definition 3 — (Regular expression over I)

(a) The symbol ϕ is a regular expression; the symbol λ (used for the empty string) is a regular expression.

(b) The symbol i for any $i \in I$ is a regular expression.

(c) If A and B are regular expressions, then (AB), $(A \vee B)$, and $(A)^*$ are regular expressions.

Definition 4 — (Regular sets) Any set represented by a regular expression according to the conventions described below is a regular set:

ϕ represents the empty set,

λ represents the set $\{\lambda\}$ containing the empty string,

i represents the set $\{i\}$.

For regular expressions A and B,

(AB) represent the set of all elements of the form $\alpha\beta$ where α belongs to the set represented by A and β belongs to the set represented by B.

$(A \vee B)$ represents the union of A's set and B's set.

$(A)^*$ represents the concatenation of members of A's set.

We note that λ, the empty string, is a member of the set represented by A^*. In writing regular expressions, we eliminate parentheses when no ambiguity results. We will also be a little sloppy and say things like "The regular set $0^* \vee 10$" instead of "The set represented by the regular expression $0^* \vee 10$."

Example 3 — Here we give some regular expressions and describe the set each one represents.

(a)$1^*0(01)^*$	(a′) Any number (including none) of 1s, followed by a single 0, followed by any number (including none) of 01 pairs.
(b) $0 \vee 1^*$	(b′) A single 0 or any number (including none) of 1s.
(c) $(0 \vee 1)^*$	(c′) Any string of 0s and 1s, including λ.

(d) $11((10)^*11)^*\ (00^*)$ (d') A nonempty string of pairs of 1s interspersed with any number (including none) of 10 pairs, this string followed by at least one 0.

We have introduced regular sets because, as we will see, these are exactly the sets finite-state machines are capable of recognizing. Thus, any set recognized by a finite-state machine is regular; and conversely, any regular set can be recognized by a finite-state machine. This result was first proved in 1956 by Stephen Kleene. First, we show that any set recognized by a finite-state machine is regular.

We represent finite-state machines by directed graphs. Temporarily, we enlarge the set of machines to include structures whose graphs may not have a full complement of arrows, so that some states under a given input symbol may have no next state defined. If we call such structures Machines (with a capital M), then a (finite-state) machine is a special case of a Machine. Although we are ultimately interested in the set of strings taking a given machine from its starting state to any final state, we first consider the set of strings taking a Machine from any one state to another, not necessarily different, state. By using induction on the size of the Machine, we prove that such a set is regular.

For the base step, assume we have a Machine with only one state, s_0. Let $K = \{i_1, i_2, \ldots, i_k\}$ be the set of input symbols for which the next-state function on s_0 is defined. We want to find a regular expression for the set of all strings taking M from s_0 to s_0. Since there is nowhere else to go, any input from K^* does this. Thus, the regular expression is $(i_1 v\, i_2 v \ldots v\, i_k)^*$. Note that the set includes λ, which certainly takes M from s_0 to s_0.

Now, we assume that in any k-state Machine, the set of strings taking the Machine from any state s_m to any state s_n is regular. Finally, we let M be a Machine with $(K + 1)$ states, and we let s_m and s_n be states in M. We consider the two cases $s_m = s_n$ and $s_m \neq s_n$.

For the case $s_m = s_n$, we first consider nonempty strings taking M from s_m back to s_m for the first time. Such strings will be of two types:

- a single input symbol $i \in I$; and
- a string of the form $i_p \alpha i_q$ where i_p, $i_q \in I$. i_p moves M from s_m to a different state s_{m1}. α is a string moving M from s_{m1} to some, not necessarily different, state s_{1m} but keeping it away from s_m; and then i_q takes M from s_{1m} back to s_m.

Let A be the set of all input strings taking M from s_{m1} to s_{1m} without going through s_m. If we delete s_m, the rest of the Machine is a K-state Machine, and A is regular by the induction hypothesis. For a fixed i_p and i_q, $i_p A i_q$ is a regular set. The set B of all strings of the form (2), above, is the union of a finite number of such sets (taking the union over the various $i_p s$ and $i_q s$); hence, B is regular. And the set C of all strings previously described is the union of B with a finite number of single input symbols; thus C is also regular. Now C^* denotes the set of concatenations of members of C and describes the set of all input strings taking M from s_m to s_m; C^* is regular.

Now we need to handle the second case where $s_m \neq s_n$. Again, we first consider the set E of all strings moving M from s_m to s_n for the first time. Any such string is of the form αi where α takes M from s_m to some $s_{1m} \neq s_n$ but keeps it away from s_n, and i takes M from s_{1m} to s_n. Let D be the set of ail input strings taking M from s_m to s_{1m} without going through s_n. If we disconnect s_n, the rest of the Machine is a K-state Machine, and so D is regular by induction hypothesis. For a fixed i, Di is therefore a regular set. The set E then consists of the union of a finite number of such sets (taking the union over various is); E is also regular. Now let F denote the set of all strings taking M from

s_n to s_n: we know F is regular by the previous case. The regular set EF is then the set of all input strings taking M from s_m to s_n.

We have shown that the set of input strings taking a Machine M from any one state to any one state is regular. The set of strings taking a (finite-state) machine M from s_0 to any final state is the union of a finite number of such sets, and so it is regular. On the other hand, if M has no final states, the empty set ϕ is the only set "recognized," and ϕ is also regular.

We have proved the first half of:

Kleene's Theorem (Part 1)

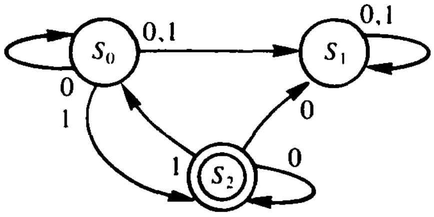
Any set recognized by a finite-state machine is regular.

This theorem states that given a finite-state machine M, there exists a regular expression describing the set of strings M recognizes. The proof, however, does not tell us how to find such an expression.

The other half of the Kleene theorem states that for any regular set, there is a finite-state machine recognizing it. To prove this result, we will introduce a new kind of machine called a *nondeterministic finite-state machine*. This machine is defined as an ordinary finite-state machine except that for each state-input pair, the next state need not be uniquely determined and there is, in fact, a set of possible next states; this set could even be ϕ. In other words, the state function f_s maps $S \times I$ to the set of subsets of S.

Example 4 — Here is the state table and state graph of a nondeterministic machine M.

Present State	Next State		Output
	Present 0	Input 1	
s_0	s_0, s_1	s_1, s_2	0
s_1	s_1	s_1	0
s_2	s_1, s_2	s_0	1



As a nondeterministic machine acts upon an input string α, the first input symbol processed leads M from the starting state to a set of possible next states. Each of these states, upon processing the second symbol, has a set of possible next states; the union of these sets is the set of possible states for M after processing two symbols of α. If we continue this procedure, we find the set of possible states for M after processing α. If any of the states in this set is a final state of M, we say that M *recognizes* α. The set of strings so recognized is the set recognized by M.

Example 4 (continued) — The nondeterministic machine of Example 4 recognizes the set 0*1(OV10*1)*. For any string α in this set, M has a possible sequence of moves that would result in M being in a final state at the end of processing α.

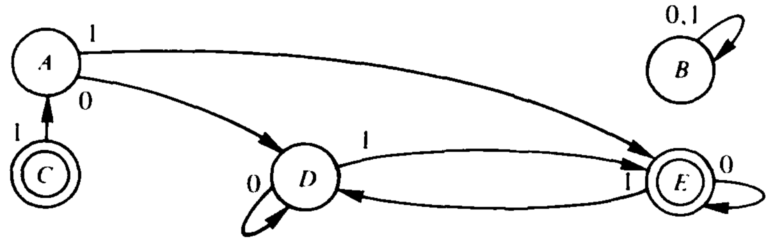
A nondeterministic machine M does not operate by choosing at each clock pulse some next state out of a set of possible next states. Rather, it operates like a parallel processor, keeping track at all times of all its possible configurations. , we simulate M's behavior by running in parallel a bunch of deterministic machines, each of which traces out a different possible sequence of moves for M. We can also simulate M's behavior by constructing a single big deterministic machine with enough states to represent all of M's possible configurations.

Lemma — For any nondeterministic machine M recognizing a set S, there is a deterministic machine M' also recognizing S.

Proof — The states of M' are sets of states of M. If s_0 is the starting state of M, then $\{s_0\}$ is the starting state for M'. For each state $\{s_{i_1}, \ldots, s_{i_k}\}$ of M' and each input symbol i, we find the next state of M' by taking the union of the set of next states for s_{i_1}, under i in M, s_{i_2} under i in M, etc. A state of M' is labeled a final state if and only if it contains a final state of M.

Example 4 (continued) — Here we give the deterministic counterpart of the nondeterministic machine of Example 4. The state table and graph follow:

Present State	Next State		Output
	Present 0	Input 1	
A = $\{s_0\}$	$\{s_0, s_1\}$	$\{s_1, s_2\}$	0
B = $\{s_1\}$	$\{s_1\}$	$\{s_1\}$	0
C = $\{s_2\}$	$\{s_1, s_2\}$	$\{s_0\}$	1
D = $\{s_0, s_1\}$	$\{s_0, s_1\}$	$\{s_1, s_2\}$	0
E = $\{s_1, s_2\}$	$\{s_1, s_2\}$	$\{s_0, s_1\}$	1

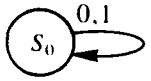


For example, the next state of $\{s_1, s_2\}$ under 1 is $\{s_0, s_1\}$ because the set of next states in M for s_1 under 1 is $\{s_1\}$ and the set of next states in M for s_2 under 1 is $\{s_0\}$. From the state graph for M', we see that M' recognizes 0*1(0V10*1)*; we also see that states B and C are "unreachable" from the starting state A and so could be eliminated.

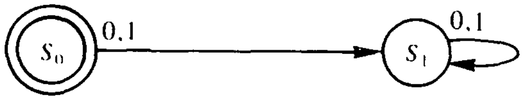
In our example, the number of states in the deterministic machine M' is close to the number of states in the original non-deterministic machine M. This situation may not be; if M has n states, M' could have as many as $2^n - 1$ states.

Our Lemma says that we gain no recognition capabilities by considering nondeterministic machines. Therefore, the proof of Kleene's theorem will be complete if we show that for any regular set, there is a nondeterministic finite-state machine recognizing it. We prove that such a machine exists by showing how to construct it. Because the definition of a regular expression is inductive, we must construct our machine inductively. We let I be the set of symbols, and consider various types of regular expressions.

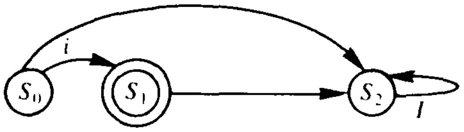
(1) ϕ and λ. A trivial machine with a single, nonfinal state, as below recognizes ϕ.



A deterministic machine that recognizes λ is



(2) For $i \in I$ a deterministic machine that recognizes i is $I - \{i\}$



(Note in (1) and (2), a deterministic machine is a special case of a nondeterministic machine.)

We now assume that for regular expressions A and B, there are nondeterministic recognizers M_A and M_B. To avoid mixups, we'll also assume the states in M_A and the states in M_B have different names.

(3) AB: The basic idea here is to connect the two machines M_A and M_B in series to create a machine M_{AB} recognizing AB. The set of states for M_{AB} is the union of the sets of states of M_A and M_B. The starting state for M_{AB} is the starting state of M_A and the final states of M_{AB} are the final states of M_B. Whenever a state-input in M_A could take M_A to a final state, we want to allow the possibility of jumping instead to the starting state of M_B, so that we begin to process strings $\beta \in B$ in M_B. Hence, we modify the state table for M_A so that whenever the set of next states contains a final state of M_A, we add the starting state of M_B to the set. Then for any $\alpha\beta \in AB$, there is a sequence of moves taking M_{AB} from its starting state through the actions of M_A on α and to the point of recognition, then transferring to perform the actions of M_B on β until β is recognized by M_B; hence, $\alpha\beta$ is recognized by M_{AB}.

(4) $A \vee B$: The basic idea here is to connect the two machines M_A and M_B in parallel to create a machine $M_{A \vee B}$ recognizing $A \vee B$. The states of $M_{A \vee B}$ are the states of M_A plus the states of M_B plus one additional state, $\bar{s}$, designated as the starting state for $M_{A \vee B}$. The final states of $M_{A \vee B}$ are the final states of M_A plus the final states of M_B. When we process the first symbol i of a string γ, we want to allow the possibility of simulating either M_A's actions in processing i beginning in its starting state s_A or M_B's actions in processing i beginning in its starting state s_B. We define the set of next states for $\bar{s}$ under i to be the union of the set of next states s_A under i and the set of next states of s_B under i. Thus, $M_{A \vee B}$ processes γ by simulating either M_A or M_B, recognizing γ if it is recognized by either M_A or M_B.

(5) A^*: $M_A{}^*$ uses the set of states of M_A plus an additional starting state $\bar{s}$, which also must be a final state in order to recognize λ. The final states of M_A are also final states of $M_A{}^*$. If i is the first symbol of a string γ, then $M_A{}^*$ should simulate M_A's actions in processing i beginning in its starting state s_A. Thus we let the set of next states of $\bar{s}$ under i be the set of next states of s_A under

i. If an initial segment of γ is recognized by A, we need to be able to reinitialize at once. Hence we modify M_A so that the set of next states for any final state and input j contains the set of next states for $\bar{s}$ under j. This modification allows the first character after the initial segment to be processed just as M_A would do it starting in S_A. Thus M_A* recognizes $A*$.

We note that slight modifications of the above procedure will be required to take care of troublesome cases involving λ. To construct a machine for 1*0*, for example, we want to leave the starting state for the machine of 1* as a final state, even though according to (3) only the final states of 0* should remain final. Similarly, a machine for 1*0 would call for a transfer on 0 from the starting state of the machine for 1* to the final state of the machine for 0.

The previous procedure should be viewed as a canonical procedure; i.e., it is completely general and always works. But for any particular case, we may be able to come up with a much simpler machine. In summary, we have proven:

Kleene's Theorem — A set is regular if and only if it is recognized by some finite-state machine.

This theorem outlines the limitations as well as the capabilities of finite-state machines, as there are certainly many sets that are not regular; e.g., $S = \{0^n 1^n \mid n \geq 0\}$ is not regular where a^n stands for a string of n copies of a. (Notice that 0*1* does not do the job.) By Kleene's theorem, there is no finite-state machine capable of recognizing S. Yet S seems like such a "nice" set, and surely you or I could count a string of 0s followed by 1s and see whether we had the same number of 0s as 1s. This lapse seems to suggest some deficiency in our use of a finite-state machine as a model of a computational device. We will try to remedy this in the next section.

TURING MACHINES

We use the terms "algorithm," "effective procedure," and "computational procedure" interchangeably, and we do not give a formal definition for any of them. Instead, we appeal to a common, intuitive understanding of an algorithm or effective procedure. We assume that any input to which an algorithm is to be applied has been encoded into numeric form, usually nonnegative integers, just as input for an actual digital computer program is encoded and then stored in binary form.

Recalling the set $S = \{0^n 1^n \mid n \geq 0\}$, let us try to see why no finite-state machine can recognize it. We probably consider ourselves to be finite-state machines and imagine that our brains, being composed of a large number of cells, can only take on a finite, although immensely large, number of configurations, or states. We feel, however, that if someone presented us with an arbitrarily long string of 0s followed by an arbitrarily long string of 1s, we could detect whether the number of 0s and 1s was the same.

For small strings of 0s and 1s, we could perhaps just look at the strings and decide. Thus we can tell without great effort that $000111 \in S$ and that $00011 \notin S$. However, for the string 000000000000000111111111111111, we must devise another procedure; e.g., we could count the number of 0s and when we get to the first 1, we write that number dowr (or remember it) and then we begin counting the 1s. (This is what we did mentally for smaller strings.) But we have now made use of some extra memory because when we finish counting 1s, we have to retrieve the number representing the number of 0s to make a comparison. But such "information retrieval" is what a finite-state machine cannot do; its only capacity for remembering input is to have a given input symbol and send it to a particular state. Suppose we attempt to build a finite-state machine to recognize S. We could count the number of 0s seen by having each new 0 move us to a new state of the machine. However, since the number of states of any given machine is a finite number, this plan fails if the number of 0s read in is larger than this finite number, so our machine clearly could not process $0^n 1^n$ for all n. In fact, if we think of solving this problem on an actual digital computer, we encounter the same difficulty. If we set a counter as we read in the 0s, we might get an overflow because our counter can only go so high. To process $0^n 1^n$ for arbitrarily large n requires that we have unlimited auxiliary memory available to store the value of our counter, which in practice cannot happen.

Another way we humans might consider attacking the problem of recognizing S is to wait until the entire string has been presented to us. We would then go to one end of the string and cross out a 0, go to the other end and cross out a 1, go back and forth to cross out another 0-1 pair, and continue this operation until we run out of 0s or 1s. The string belongs to S if and only if we run out of both at the same time. Although this approach sounds rather different from the first one, it still requires remembering each of the inputs in that we must go back and read them once the string is complete. A finite-state machine cannot reread input.

We have come up with two computational procedures to decide, given a string of 0s and 1s, whether that string belongs to S. Both procedures required some form of additional memory unavailable in a finite-state machine. Evidently, the finite-state machine is not a model of the most general form of computational procedure.

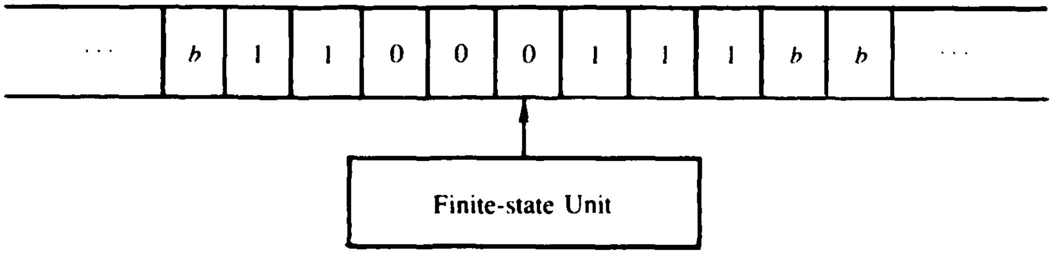
To simulate more general computational procedures than a finite-state machine can handle, we use a Turing machine (invented by A. M. Turing in 1936). A Turing machine is essentially a finite-state machine with the added ability to reread its input and also to erase and write over its input, and with unlimited auxiliary memory—thus overcoming deficiencies already noted about finite-state machines.

A Turing machine consists of a finite-state machine and a tape divided into cells, each cell containing, at most, one symbol from an allowable finite alphabet. At any one instant, only a finite number of cells on the tape are nonblank. We use the special symbol b to denote a blank cell. The finite-state unit, through its read-write head, reads one cell of the tape at any given moment (see figure below). By the next clock pulse, depending upon the present state of the unit and the symbol read, the unit either does nothing (halts) or completes three actions.

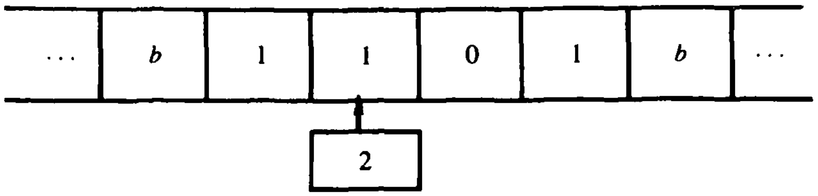
1. Print a symbol from the alphabet on the cell read (it could be the same symbol that's already there).

2. Go to the next state (it could be the same state as before).

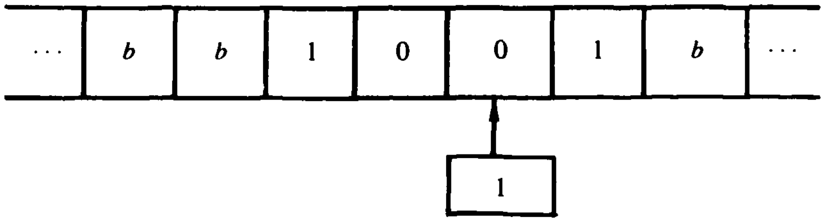
3. Move the read-write head one cell left or right.



We describe the action of any particular Turing machine by a set of quintuples of the form (s,i,i',s',d) where s and i indicate the present state and the tape symbol being read and i' denotes the symbol printed, s' denotes the new state, and d denotes the direction of the move of the read-write head (R = right, L = left). Thus a machine in the configuration



if acting according to the quintuple $(2,1,0,1,R)$ would move to the configuration



Definition 5 — Let S be a finite set of states and I a finite set of tape symbols (the *tape alphabet*) including a special symbol b. A *Turing machine* is a set of quintuples of the form (s,i,i',s',d') where $s,s' \in S, i,i', \in I$, and $d \in \{R,L\}$ and where no two quintuples begin with the same s and i symbols.

The restriction that no two quintuples begin with the same s and i symbols ensures that the action of the Turing machine is deterministic and completely specified by its present state and symbol read. If a Turing machine gets into a configuration for which its present state and symbol read are not the first two symbols of any quintuple, the machine halts.

Just as in the case of ordinary finite-state machines, we specify a fixed starting state, denoted by 0, in which the machine begins any computation. We also assume an initial configuration for the read-write head, namely, that it is positioned over the farthest left nonblank symbol on the tape. (If the tape is initially all blank, the read-write head can be positioned anywhere to start.)

The tape serves as a memory for a Turing machine, and, in general, the machine can reread cells of the tape. It can also write on the tape; therefore, the nonblank portion of the tape can be as long as desired, although there are still only a finite number of nonblank cells at any time. Hence the machine has available an unbounded, although finite, amount of storage. The limitations of finite-state machines observed earlier do not exist for Turing machines, so Turing machines should have considerably higher capabilities than finite-state machines. In fact, a finite-state machine is a very special case of a Turing machine, one that always prints the old symbol on the cell read, always moves to the right and halts on the symbol b. (A Turing machine may fail to halt, e.g., by endlessly cycling or by moving forever along the tape.)

Turing machines are usually used to do two kinds of jobs. First, they can be used as recognizers and second to compute functions. Here we only discuss their role as recognizers, much as we considered finite-state machines as recognizers. We give a similar definition, provided we first define a final state for a Turing machine. A *final state* in a Turing machine is one that is not the first symbol in any quintuple. Thus upon entering a final state, whatever the symbol read, the Turing machine halts.

Definition 6 — A Turing machine T with tape alphabet I recognizes (accepts) a subset S of I^* if T, beginning in standard initial configuration on a tape containing a string α of tape symbols, halts in a final state if and only if $\alpha \in S$.

Note that Definition 6 leaves open two alternative behaviors for T when applied to a string α of tape symbols not in S. T may halt in a non-final state or T may fail to halt at all.

We now build a Turing machine to recognize $S = \{0^n 1^n \mid n \geq 0\}$. The machine is based on our second approach of sweeping back and forth crossing out 0, 1 - pairs.

Example 5 — Here we build a Turing machine that will recognize $S = \{0^n 1^n \mid n \geq 0\}$. We will use one additional special symbol, call it X; so the tape alphabet $I = \{0, 1, b, X\}$. State 6 is the only final state. The quintuples making up T follow with a description of their function.

$(0, b, b, 6, R)$	Recognizes the empty tape (which is in S).
$(0, 0, X, 1, R)$	Erases the left most 0 and begins to move right.
$(1, 0, 0, 1, R)$ $(1, 1, 1, 1, R)$ $(1, X, X, 2, L)$ $(1, b, b, 2, L)$	Moves right in state 1 until it reaches the end of the string; then moves left in state 2.
$(2, 1, X, 3, L)$	Erases the rightmost 1 and begins to move left.
$(3, 1, 1, 3, L)$	Moves left over 1s.
$(3, 0, 0, 4, L)$	Goes to state 4 if more 0s are left.
$(3, X, X, 5, R)$	Goes to state 5 if no more 0s in string.
$(4, 0, 0, 4, L)$	Moves left over 0s.
$(4, X, X, 0, R)$	Finds left end of string and begins sweep again.
$(5, X, X, 6, R)$	No more 1s in string, machine accepts.

Is the Turing machine a better model of an effective procedure than the finite-state machine? Although our concept of effective procedure is an intuitive one, we are quite likely to agree that any procedure computable by a Turing machine is an effective procedure or algorithm. In fact, the set of quintuples of T is itself the algorithm; as a finite list of finite instructions that can be carried out mechanically, it satisfies the various conditions which could be common to anyone's notion of an algorithm. Given the simplicity of the Turing machine definition, it is startling, however, to assert that anything computable by an effective procedure can also be processed by means of a Turing machine. This is the statement of the:

Church-Turing Thesis — Any process which could naturally be called an effective procedure can be realized by a Turing machine.

Because the Church-Turing thesis equates an intuitive idea (effective procedure or algorithm) with a mathematically precise, well-defined idea (the Turing machine), it can never be formally proved and must remain a "thesis," not a "theorem." What, then is its justification?

One piece of evidence is that whenever a procedure everyone could agree was an effective procedure (according to his or her own insights into this idea) has been proposed to compute something, someone has designed a Turing machine to also do the computation. (Of course there is always the nagging thought that someday this might not happen.)

Another piece of evidence is that other mathematicians, several of them at about the same time that Turing developed the Turing machine—late 1930s or early 1940s—also proposed models of effective procedures. On the surface, each proposed model seems unrelated to any of the others. Because all of these models are formally defined, just as a Turing machine, it is possible to consider on a formal, mathematical basis whether any of them are equivalent. These models, as well as the Turing machine, have been proven equivalent; i.e., they define the same class of functions which suggests that Turing computability embodies everyone's concept of effective procedure.

The Church-Turing thesis is now widely accepted as a working tool in research in the area of complexity. By accepting this thesis, we have accepted the Turing machine as the ultimate model of an effective computational device. Its capabilities exceed those of any actual computer that, after all, does not have the unlimited tape storage of a Turing machine. It is remarkable that Turing proposed this concept in 1936, well before the advent of the modern computer.

COMPUTATIONAL COMPLEXITY

Suppose we have an algorithm A solving problem P. Here we may be thinking of an algorithm as a Turing machine or as an actual computer program. In either case, we are interested in how fast our algorithm works. Is it possible to devise an algorithm A′ to solve P that is "faster" (more efficient) than algorithm A, if the number of A′ basic operations is smaller than the number of A basic operations? Of course, A′ and A must have comparable basic operations, and we must be comparing A′ and A in the same environment; e.g., we cannot compare the number of steps in a Turing machine computation with the number of steps in a computation done in some higher level programming language. We use Turing machine computations as our environment; by the Church-Turing thesis, we express any algorithm as a Turing machine computation. There are other models of computation that could be used, such as a radom access machine (RAM). A RAM works somewhat like a Turing machine, but its allowable operations resemble more closely those in actual programming languages—there are arithmetic operations, branching instructions, etc.

The efficiency of an algorithm is also known as its *complexity* — this is merely some sort of measure as to the amount of work the algorithm must do. A straightforward algorithm in its logic may still require a large number of steps to carry out on a Turing machine, for example.

Suppose that the set S is recognized by a Turing machine T. We will only consider cases where T halts on all inputs since we want to count the number of steps in T's computation of S, and we don't want T to go on indefinitely. As T does a computation, we encode the input in some way on T's tape, start T in standard initial configuration, and count the number of steps (clock pulses) in the computation until T halts. We would expect that the number of steps required for T to process any $\alpha \in S$ (or $\alpha \notin S$) would be a function of the length of the input.

Definition 7 — Let T be a Turing machine that halts on all inputs. If the maximum number of steps for a computation by T on any input of length n is $t(n)$, then T is of *time complexity* $t(n)$.

Example 6 — Consider the Turing machine of Example 5 that recognizes the set $S = \{0^n 1^n \mid n \geq 0\}$. The maximum number of steps in a computation occurs when the input $\alpha \in S$. Suppose an input of length n is a member of S. The computation first moves the read-write head beyond the right end of the input, which requires n steps. The read-write head then sweeps back and forth across that part of the input not replaced by Xs. This process requires successively $n, n-1, n-2, \ldots, 1$ steps. Recognition requires one final step. Thus the total number of steps is

$$n + (n + n - 1 + \ldots + 1) + 1 = \frac{n^2 + 3n + 2}{2} .$$

This Turing machine is of time complexity $t(n) = (1/2)(n^2 + 3n + 2)$.

Another measure of efficiency, which we will not consider here, is the *space complexity* of a Turing machine, a measure of the amount of tape the machine uses as a function of input length.

Definition 8 — Let f and g be two functions from $N \rightarrow N$. Then f and g are of the same *order of magnitude* if there exist positive constants c_1 and n_1 such that $f(n) \leq c_1 g(n)$ for all $n \geq n_1$ and there exist positive constants c_2 and n_2 such that $g(n) \leq c_2 f(n)$ for all $n \geq n_2$.

We note that the algorithm of Examples 5 and 6 is of order n^2. Suppose we have two algorithms to do the same job and their time complexities are of different orders of magnitude, say A is of order n and A' of order n^2. Even if each step in a computation takes only 0.0001 s, this difference will affect total computation time as n grows large. Also suppose we have a third algorithm A'' whose time complexity is an exponential function, say 2^n. The table below compares total computation time for A, A', and A'' under various input lengths:

Algorithm	Order	Size of Input		
		10	50	100
A	n	0.001 s	0.005 s	0.01s
A'	n^2	0.01 s	0.25 s	1s
A''	2^n	0.1024 s	3570 yr	4 × 10^{16} centuries

Because of the immense growth rate of algorithms not of polynomial order, these are not useful for large values of n. In fact, problems for which no polynomial time algorithms exist are called *intractable*. There may, however, be extenuating circumstances. When an algorithm has time complexity $t(n) = 2^n$, say, at least one input of length n requires 2^n steps, but the average case may run much faster. In general, however, a choice between possible algorithms for a given problem, or attempts to improve a given algorithm, should concentrate on the order of magnitude of the time complexity functions involved.

Definition 9 — **P** is the collection of all sets recognizable by Turing machine of polynomial time complexity (**P** stands for polynomial time).

Consideration of set recognition in Definition 9 is not as restrictive as it may seem. Since a Turing machine for which a time complexity can be determined must halt on all inputs, it decides membership in a set. Furthermore, many problems can be posed as set decision problems through a suitable encoding of the objects involved in the problem. The particular encoding scheme we use determines the length of the input string for a given instance of a problem, and thus may affect the time complexity of an algorithm to solve the problem. However, if there are two encodings for a given problem, such that inputs under each encoding can be transformed in polynomial time to corresponding inputs under the other encoding, then if one encoding results in a set belonging to **P**, so does the other.

The situation is equally pleasant with respect to alternative models of algorithms (Turing machine, RAM's etc.). A problem solvable by an algorithm with polynomial time complexity on one model is solvable by an algorithm with polynomial time on another model. Thus we speak of a *problem* belonging to **P**, meaning that a polynomial time-bounded algorithm exists for its solution, without having to specify the computational device that carries out the algorithm or the details of the encoding problem for that device (e.g., we could ask does the Hamiltonian circuit problem belong to **P**, but no one yet knows the answer — this asks if an arbitrary graph has a cycle using every vertix of the graph.)

The decision problem for Hamiltonian circuits, unlike problems such as the Halting problem in Complexity theory (Does an algorithm exist to decide, given a Turing machine T and a string α, whether T begun on a tape containing α will eventually halt?) and the word problem in Group theory (Does an algorithm exist to decide, given the generators and defining relators—a presentation—for a group and a word from the group, whether the word can be transformed to the identity?), is *not* unsolvable. An algorithm exists to test whether an arbitrary graph has a Hamiltonian circuit, viz, the trial-and-error approach of testing all possible paths. The same thing is true of the factoring problem. We can simulate this type of behavior by using a *nondeterministic Turing* machine (NDTM). An NDTM is defined just like an ordinary Turing machine except that for each state-input pair, there is a set of applicable quintuples and so, possibly, a choice for the Turing machine's behavior at that point. (This corresponds to the situation, e.g., with the Hamiltonian circuit problem that as we trace out paths, we may have a choice of possible next moves every time we come to a vertix of the graph.) Each choice (each quintuple) specifies the symbol to be printed, the next state, and the direction of motion of the read-head. We think of the NDTM as pursuing all of its possible sequences of action in parallel. An NDTM T *recognizes*, or *accepts*, a string α of tape symbols if T, begun in standard configuration on α, has some sequence of moves leading to a halt in a final state. T recognizes the set of all recognized strings.

Definition 10 — Let T be an NDTM. For every recognized input string α of length n, there is at least one sequence of moves leading to a final state; for each accepted string, consider only the shortest sequence of moves leading to acceptance. If the maximum number of steps used in any such sequence accepting a string of length n is $t(n)$, then T is of *time complexity* $t(n)$.

Definition 11 — **NP** is the collection of all sets recognizable by NDTMs of polynomial time complexity. (**NP** stands for *nondeterministic* polynomial time.)

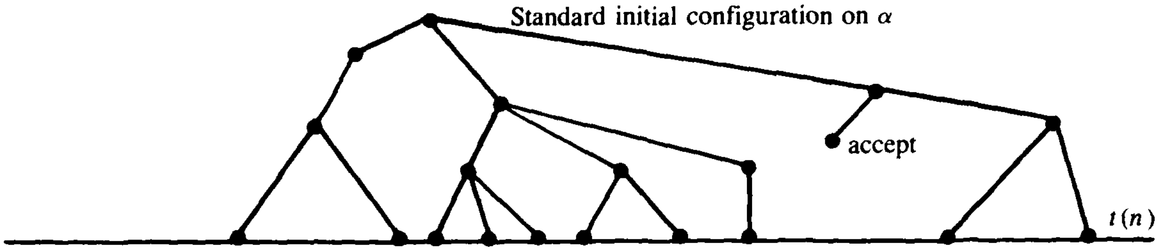
Any ordinary (deterministic) Turing machine is a trivial NDTM, so it is clear that $\mathbf{P} \leq \mathbf{NP}$. Whether **P** is a proper subset of **NP** is the question which occupies us for the rest of this section.

As in the corresponding case of finite-state machines (see our Lemma in the proof of Kleene's theorem), any set recognizable by an NDTM T can also be recognized by a deterministic Turing machine T'. We can think of T' acting on a given input α as simulating one after another the possible sequences of moves T could make on α until α is accepted or all possible sequences have been tried and α is rejected. Therefore, although nondeterminism gains us no new capabilities, we would expect it to gain us some lower time complexity.

Thus if the time complexity for T is $t(n)$, we would expect the time complexity $t(n)$ for T' to be higher for two reasons. T' cannot execute sequences of moves in parallel as T can; it must do them in a serial fashion. Also, T' gives us more information about an input α of length n; although T may give an answer within $t(n)$ units of time only if α is accepted, T' always gives us an answer (yes or no) about any input α within $t'(n)$ units of time. There is one detail we glossed over in discussing T''s simulation of T on α. T may have sequences of moves that do not halt; if T' begins simulating one of these sequences, how does it know when to give up and try another sequence? If T has time complexity $t(n)$, then T' need not pursue any sequence of moves for longer than $t(n)$ units of time. If α is accepted by T, there is some sequence that will do the job within this time. We may imagine T''s possible actions on a given α as something like the tree shown in the figure below; as T' simulates T, it need not look below $t(n)$ levels, and it can trace out each branch of the tree that far. Because there is a bound b on the maximum number of possible moves T can make at any point, there are at most b branches of the tree at any vertix. Thus the tree can have at most $b^{t(n)}$ separate paths, each of length at most $t(n)$, so we would expect some exponential expression such as

$$t(n)b^{t(n)}$$

to be the time complexity for T'. We should never need more time than this, but probably some input of length n for some n might require this much time.



The previous argument seems to prove that, in most cases, if a set is accepted by an NDTM of time complexity $t(n)$, it will probably require a deterministic machine of time complexity that looks like $t(n)b^{t(n)}$, a function of a higher order of magnitude. Such a result has not been proven, however. No one has found any set S recognizable by an NDTM with time complexity $t(n)$ for which no deterministic machine of complexity $t(n)$ exists to recognize S. Although there are certainly sets for which such a deterministic machine has not been found, it has not been established that one cannot exist. In particular, whether **P** is a proper subset of **NP** is an open question.

There are many famous problems such as the Hamiltonian circuit problem, the factoring problem, and the knapsack problem that have been shown to be in **NP**, i.e., they are representable as **NP** sets, but for which no polynomial-bounded, deterministic solution algorithm has been found. This fact lends weight to the speculation that **P** is indeed a proper subset of **NP**. This view is the prevailing one in complexity theory circles today. It is strengthened by work begun in 1971 on a class of problems known as **NP**-complete (**NPC**) problems. Roughly, if a problem is **NPC**, it is **NP** and at least as hard to solve as any other **NP** problem in that if it could be shown to belong to **P**, then every **NP** problem would belong to **P** and **P** would equal **NP**.

Many problems from different fields (graph theory, number theory, etc.) have been shown to be **NPC**. For example, both the Hamiltonian circuit problem and the knapsack problem are **NPC**. The **NPC** problems are diverse, and the search for efficient (polynomially bounded) solution procedures has been extensive. In view of the so far unsuccessful search for an efficient solution procedure for even one such problem, it seems likely that $\mathbf{P} \neq \mathbf{NP}$. On the practical side, however, one should not look too long for a quick and easy algorithm to solve any **NP** problem one may encounter.

CONCLUSION

The subject of complexity theory deals with the following two aspects of any problem: the most efficient method of obtaining a solution and the number of operations needed to perform this task. The idea of using intractable problems in the design of cryptosystems seems to be attractive; however, there are a number of difficulties with this. Shamir [3] has pointed out:

(1) Complexity theory deals with only the worst possible case of any problem. It could be that only very few instances of a problem are truly intractable. A cryptosystem based on such a problem would only occasionally be secure.

(2) Complexity theorists assume that only a certain amount of information is available for the solution of an instance of a problem. Cryptoanalysts frequently have much more information at their disposal, such as corresponding plaintext and ciphertext.

(3) Given any particular difficult problem, it is not always possible to convert it into a cryptosystem. As a matter of fact, most known public key techniques are based on only two NP-problems: the factoring problem and the knapsack problem.

Moreover it has been conjectured that the breaking of any public key cryptosystem is *not* as hard as an **NPC** problem. A piece of evidence which supports this is the breaking of the Merkle-Hellman trapdoor Knapsack cryptosystem [3], despite the fact that the knapsack problem is itself **NPC**. Furthermore, it would be very desirable to have a proof of the equivalence of the problem of breaking the RSA system and the factoring problem. At this time, no such proof exists. Thus the present state of complexity theory is inadequate to demonstrate the computational infeasibility of any cryptosystem. What is needed are new measures of complexity especially tailored to the problem of cryptoanalysis. Admittedly, while this appears to be a very difficult mathematical problem, it would be worthwhile pursuing it because when we can certify the security of cryptosystems according to such measures of cryptocomplexity, the problem of secure communications will be solved.

REFERENCES

1. R. Rivest, A. Shamir, and L. Adleman, "A Method for Obtaining Digital Signatures and Public-key Cryptosystems," *Comm. ACM*, **26** 96-99 (1983).

2. H. Griffin, *Elementary Theory of Numbers* (McGraw-Hill, New York, NY, 1954).

3. A. Shamir, "A Polynomial Time Algorithm for Breaking Merkle-Hellman Cryptosystems," (extended abstract), the Weizmann Institute, Israel (1982).

BIBLIOGRAPHY

1. Aho, A.V., J.E. Hopcroft, and J.D. Ullman, *The Design and Analysis of Computer Algorithms* (Addison-Wesley, Reading, MA, 1974).

2. Garey, M.R. and D.S. Johnson, *Computers and Intractability, A Guide to the Theory of NP-Completeness* (W.H. Freeman, San Francisco, 1979).

3. Hopcroft, J.E., and J.D. Ullman, *Introduction to Automata Theory, Languages, and Computation* (Addison-Wesley, Reading, MA, 1979).

4. Minsky, M.L., *Computation: Finite and Infinite Machines* (Prentice Hall, Englewood Cliffs, NJ, 1967).

5. Williams, H.C., "Computationally 'Hard' Problems as a Source for Cryptosystems," *Secure Communications and Asymmetric Cryptosystems*, G.J. Simmons, ed., AAAS Selected Symposium 69, 11-39 (1982).